Eyu Chen

CS124_project5

Dec 3

Project5 Report

In this project, we are attempting to use hash table to store the dataset. Two kinds of hash tables would be used in this project – separate chaining and quadratic probing. First, we will use default setting of hash function and key getter, and try with 5 different sizes to figure out the best size that we could use for the following operations. Since we get the proper size, we can work with hash table further on different hash function and key getter. In addition, we will record the time of each insertion.

Dataset

Data The data file used is about all the fall courses in University of Vermont in 2021, the source is from the Office of Register https://serval.uvm.edu/~rgweb/batch/curr_enroll_fall.html, and was modified a little bit.

There are 9 attributes in the .csv file and then become the fields for the class, some naming follows abbreviations used in CSV file:

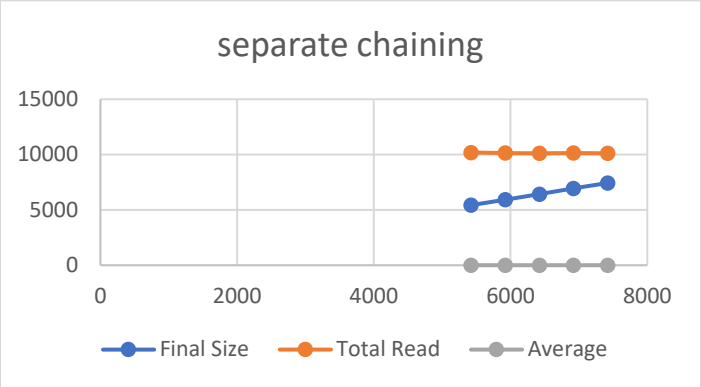
- rowId: the row ID for ordering
- subject: the subject that the course belongs to
- title: the title of the course
- CRN: the registration number for this course, all of CRN are in 5 digits. There are some classes share the same CRN, and this field will be used to be the default key getter (int to string)
- college: the college code that the class belongs to
- maxEnrollment: the maximum enrollment of the course
- currEnrollment: the current enrollment of this course by now
- instructor: the professor's name of the course, in the format of FirstName, LastName
- email: the uvm email address of the professor, this field will also be used to be a different key getter

Best Size

At the beginning, we use various table sizes to find the best one for the dataset. The size of data is 5422, and is increased by 500 in 4 times:

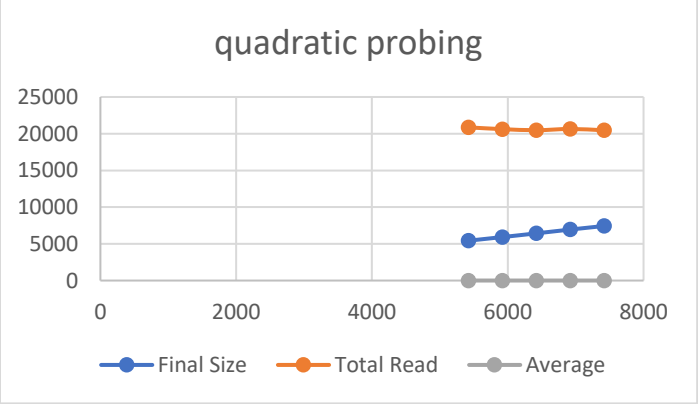
1. Separate Chaining

Initial Size	Final Size	Total Read	Average
5422	5431	10191	1.87645
5922	5923	10141	1.71214
6422	6427	10117	1.57414
6922	6947	10138	1.45933
7422	7433	10113	1.36055



2. Quadratic Probing

Initial Size	Final Size	Total Read	Average
5422	5431	20860	3.84091
5922	5923	20623	3.48185
6422	6427	20476	3.18593
6922	6947	20644	2.97164
7422	7433	20461	2.75272



To select the best size, we focus on the lowest difference between the initial size and final size. Therefore, we choose 5922 as the best size as it only increases the size by 1 in both hashing cases. The results were outputted in the terminal window as the follow:

Default table in size 5422

Total # of Separate chaining: 10191

Final size of Separate chaining: 5431

Timer: 15620 microseconds

Total # of Quadratic probing: 20860

Final size of Quadratic probing: 5431

Timer: 31243 microseconds

++

Default table in size 5922

Total # of Separate chaining: 10141

Final size of Separate chaining: 5923

Timer: 140021 microseconds

Total # of Quadratic probing: 20623

Final size of Quadratic probing: 5923

Timer: 30126 microseconds

++

Default table in size 6422

Total # of Separate chaining: 10117

Final size of Separate chaining: 6427

Timer: 20087 microseconds

Total # of Quadratic probing: 20476

Final size of Quadratic probing: 6427

Timer: 20088 microseconds

++

Default table in size 6922

Total # of Separate chaining: 10138

Final size of Separate chaining: 6947

Timer: 20102 microseconds

Total # of Quadratic probing: 20644

Final size of Quadratic probing: 6947

Timer: 20093 microseconds

++

Default table in size 7422

Total # of Separate chaining: 10113

Final size of Separate chaining: 7433

Timer: 20100 microseconds

Total # of Quadratic probing: 20461

Final size of Quadratic probing: 7433

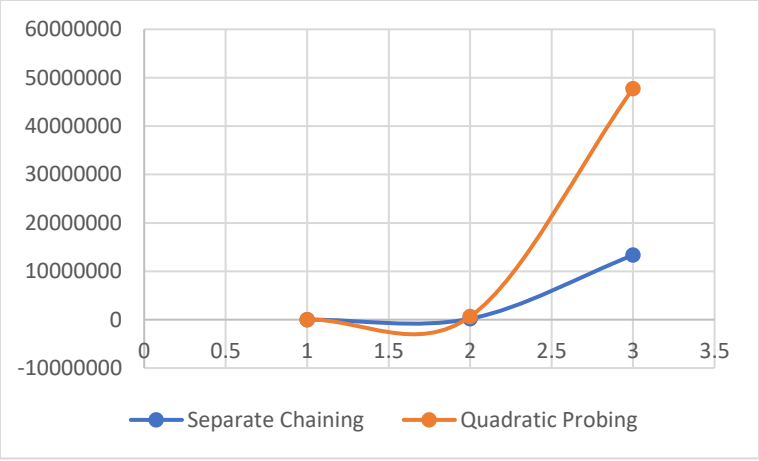
Timer: 20179 microseconds

++

Analysis on 3 Special Cases

With achieving the best size of our dataset (which is 5922), we are now working on different hash and getKey functions. In default, we use the email information of each object as a key, and the Horner function to hash the item; also we have another two functions for getting key and hashing. intToString() method is to switch the CRN data of each item to string and turns out to be a key; while hashHash() method is a hash function that simply calculates the modulus of the length of string and the table size.

	Separate Chaining	Quadratic Probing
Different getKey	6961	25314
Different hash function	194381	633319
Both are Different	13324877	47767260



the 3 points on each line represents the 3 cases.

Clearly when we change the hash function to that simpler one, the number of reads increases dramatically, and the reason should be easy to determine. For the dataset, it is possible to have the same email when using it as key, but not often; however, the issue raised when using CRN as a key as CRN is a code that is in 5 digits and some items share the same CRN, and that might cause a number of the same keys which make the program to do more and more re-indexing operations. Also, a bad hash function can exacerbate the issue, which makes the number of reads increases in crazy. In conclusion, the Horner function and the email field would be appropriate for out dataset.

Questions

1. *Consider how removing and searching compares with inserting, in terms of number of hash elements read. Would they read more, less, or the same?*
   Searching functions are usually used in both inserting and removing, and it reads item only when the item is found, thus the searching should read less than removing and inserting; removing should return a similar number of reads to inserting function.

2. *How often would each of the three operations (inserting, removing, searching) typically be used with your data set? Which hash collision method works best for your data set? Why?*
   As one step of removing and inserting, searching operation should be used frequently in running. Only in this project, there is no need to remove any items but we have to insert all. Comparing with the data, separate chaining should be a good choice for our dataset.

Timer

In this project, the timer also be used to record the time spent on the insertion in each case:

1. Hash tables (separate chaining & quadratic probing)

   Default table in size 5422

   Timer: 37761 microseconds

   Timer: 31243 microseconds

   ++

   Default table in size 5922

   Timer: 37757 microseconds

   Timer: 31241 microseconds

   ++

   Default table in size 6422

   Timer: 31242 microseconds

   Timer: 37881 microseconds

   ++

   Default table in size 6922

   Timer: 15620 microseconds

   Timer: 31241 microseconds

   ++

   Default table in size 7422

   Timer: 28734 microseconds

   Timer: 20178 microseconds

   ++

   Hash table with a different getKey function

   Timer: 20132 microseconds

   Timer: 30254 microseconds

   ++

   Hash table with a different hash function

   Timer: 242212 microseconds

   Timer: 271923 microseconds

   ++

   Both differ in getKey ans hash function

   Timer: 14335154 microseconds

   Timer: 23688772 microseconds

2. Sorting Algorithms

Bubble Sort of 5422records: 2835495microseconds

Selection Sort of 5422records: 172686microseconds

Quick Sort of 5422records: 18196microseconds

Heap Sort of 5422records: 8076microseconds

Two Sort of 5422records: 561880microseconds

Insertion Sort of 5422records: 334040microseconds

Merge Sort of 5422records: 32485microseconds